Firestorm: Operating Systems for Power-Constrained Architectures

Sankaralingam Panneerselvam and Michael M. Swift
Computer Sciences Department, University of Wisconsin–Madison
{sankarp, swift}@cs.wisc.edu

Abstract

The phenomenon of Dark Silicon has made processors over-provisioned with compute units that cannot be used at full performance without exceeding power limits. Such limits primarily exist to exercise control over heat dissipation. Current systems support mechanisms to ensure system-wide guarantees of staying within the power and thermal limit. However, these mechanisms are not sufficient to provide process-level control to ensure applications level SLAs: power may be wasted on low-priority applications while high-priority ones are throttled.

We built Firestorm, an operating system extension that introduces power and thermal awareness. Firestorm considers power a limited resource and distributes it to applications based on their importance. To control temperature, Firestorm also introduces the notion of thermal capacity as another resource that the OS manages. These abstractions, implemented in Firestorm with mechanisms and policies to distribute power and limit heat production, help applications to achieve guaranteed performance and stay within the system limits. In experiments, we show that Firestorm improved performance by up to 10% by avoiding thermal interference and can guarantee SLAs for soft real time applications in the presence of limited power and competing applications.

1. Introduction

Moore's law paved the way for doubling the transistors in the same chip area by reducing transistor sizes with every generation while also scaling voltage down. However, with the end of Dennard's scaling, voltage and hence the power draw of transistors is no longer dropping proportionally to size. As a result, modern processors cannot use all parts of the processor simultaneously without exceeding the power limit. This manifests as an increasing proportion of *dark silicon* [7]. In other words, the compute capacity of current and future processors is and will be over-provisioned with respect to the available power.

Power limits are influenced by different factors such as the capacity of power distribution infrastructure, battery supply limits, and the thermal capacity of the system. Power limits in datacenters can arise from underprovisioning power distribution units relative to peak power draw. Energy limits are also dictated by the limited capacity of batteries. However, in many systems, the primary limit comes not from the ability to acquire power, but instead from the ability to dissipate power as *heat* once it has been used.

Thermal limits are dictated by the physical properties of the processor materials and also comfort of the user—people do not want their legs scorched when sitting with a laptop. Thus, power is limited to prevent processor chips from over-heating, which can lead to thermal breakdown. As a result, the maximum performance of a system is limited by its *cooling capacity*, which determines its ability to dissipate heat. Cooling capacity varies across the computing landscape, from servers with external chilled air to desktops with large fans to laptops to fan-less mobile devices. Furthermore, cooling capacity can change dynamically with software-controlled fans [32] or physically reconfigurable systems, such as dockable tablets [33].

Processors support mechanisms to enforce both power and temperature limits. For example, recent Intel processors provide Running Average Power Limit (RAPL) counters to enforce a power limit on the entire processor [27]. In software, power capping services, such as the Linux power capping framework [25] uses these limits to control power usage. Processor vendors define a metric Thermal Design Power (TDP) for every processor model to guide the requirements of the cooling system needed to dissipate power. Most processors have a safeguard mechanism that throttles the processor by reducing the frequency or duty cycle (fraction of cycles where work happens) on reaching a critical temperature. In software, the thermal daemon [34] aims to increase performance by deploying increasing cooling (e.g., increasing fan speed) if possible before resorting to throttling.

Challenges. The drawback with current hardware and software mechanisms that enforce power and thermal limits are that they only offer system-wide guarantees but do not enable application-level guarantees.

Power distribution: When power is limited, current systems (hardware and software) throttle all applications equally. However, this approach ignores users' scheduling priorities:

power should be distributed among applications based on their importance, so that high-priority applications can use more power to run faster, while low-priority applications have their power reduced and bear most of the performance lost.

Thermal interference: An application that makes heavy use of a processor can trigger temperature throttling that reduces CPU frequency or duty cycle. This can affect all cores, and hence all running applications. Furthermore, throttling stays in effect for a while as the processor cools. As a result, a low priority or malicious application can trigger throttling that reduces the performance of high-priority applications.

Performance Boosting: Applications may be able to temporarily overclock the core to achieve extra performance if they run for short enough periods that throttling does not get triggered. This is often called computational sprinting [26]. Turbo Boost in Intel processors [12] is similar but more conservative. Such sprinting techniques require enough thermal headroom to run without overheating and also requires the processor stay cool between activities.

Prior work. There has been a great deal of prior work with respect to power and thermal management. They can be classified under the theme of improving performance (e.g., [14]) or energy efficiency (e.g., [31]), and minimizing interference within a power or thermal limit (e.g., [8]). These works differ from ours in that they assume a homogeneous set of application performance goals, while our work deliberately targets systems that run a mix of soft real time, best effort, and background tasks. Also, power and thermal capacity are dependent on each other, where one without another does not guarantee performance. Power without thermal capacity results in throttling and thermal capacity without power results in low performance. Our system takes a holistic view on both power and thermal capacity by allowing applications to co-allocate them for better performance.

Firestorm. Just as the OS actively manages resources such as CPU and memory, we argue that operating systems should treat power and thermal capacity as primary resources in the system. Towards this goal, we built Firestorm, which extends Linux with power and thermal awareness. Firestorm introduces new abstractions to manage power and thermal capacity; new interfaces are added (a) for applications to gather power and thermal resources, and allow application-specific use of power and thermal capacity for guaranteed performance; and (b) to support policies to balance the varied performance requirements of different applications.

Firestorm requires all applications to gather power before executing their tasks on any compute unit (e.g., CPU or GPU). This is enforced through agents that act as resources managers for each type of compute unit. The agent predicts the power requirement of a task using a compute-unit-specific power model and tries to acquire the required power from a centralized power center. The center employs a proportional share policy to distribute power, which isolates applications from each other. When sufficient power is not available, the agent runs a task at lower power and hence lower performance.

In order to support the thermal requirements of high-priority programs, Firestorm introduces a new *thermal conserve* policy that allows applications to reserve thermal capacity (i.e., keep the processor cooler) needed during execution. Thus, the system does not allow other applications to exhaust the thermal capacity by raising the temperature too high. Firestorm incorporates a system-specific thermal model to predict the thermal capacity needed based on the work to be performed by an application. Firestorm also extends Chameleon's [21] execution object abstraction to create thermal headroom for sequential applications. This is done by creating execution object over multiple cores where few cores are forced to an idle state and the resultant power savings is used to boost the performance of the active cores.

For cases where applications do not reserve thermal capacity, Firestorm supports a *selective throttling* policy to isolate applications from thermal interference caused by background or low priority applications. When the processor is nearing the critical temperature (temperature beyond which processors can breakdown), power given to low priority applications is reduced. This preserves the performance of important applications at the cost of low priority applications.

The contributions of Firestorm include actively allocating power to applications, reserving thermal capacity for high-priority applications, designing a set of interfaces a system requires from a power and a thermal model for making power management decisions, and simple power and thermal models that are fast and simple enough to deploy practically.

Through experiments we show that Firestorm is able to assign more power to applications with higher shares and prevent interference from lower priority programs. Firestorm also balances the performance of multiple applications under a power budget compared to the native Linux RAPL mechanism that prefers parallel programs over sequential programs. In a case where thermal interference from background application costs performance, Firestorm allows high-priority tasks to run at full speed, as compared to 19% slower under native Linux, while background tasks run 28% slower to reduce heat production.

2. Background and Motivation

2.1 Background

The power consumption of processors is usually dependent on the speed (frequency) at which it operates. Higher performance can be achieved by feeding more power to the compute units in the processor. However, the increase in power consumption results in increased heat dissipation.

Power controls. Power limits exist at a processor level to either control heat production or to limit system power to what is available from the power infrastructure (e.g., PDU capacity). Power limits can be enforced in two ways. First, by fixing the maximum frequency for the processor, a maximum bound on the power consumption can be established, although actual power can vary widely based on the workload. Second, by fixing the actual power consumption, processors can be run at any frequency as long as they stay within the power limit.

Most current processors support both mechanisms. In Intel processors, the RAPL mechanism runs cores at maximum core frequency possible and then throttles the frequency when power consumption exceeds a specified limit. Also, as their processors do not support per-core DVFS, the frequency of all cores will be reduced uniformly when throttled.

The DVFS (Dynamic Voltage Frequency Scaling) mechanism increases/decreases performance by altering the voltage and frequency level of the processor. Since the dynamic power consumption is proportional to square of voltage and frequency, power savings by reducing performance level is higher. Intel and AMD processors support different power planes for CPU and on-chip GPU, and thus the chip offers two voltage regulators for each of them. This is also why per-core DVFS is not possible, since a single regulator controls the voltage/frequency controls of all cores. On the other hand, duty cycle modulation (stopping the clock for short periods of 3μs at regular intervals) can operate at individual cores but it does not change the voltage. As a result, the power savings achieved by duty cycle modulation is below DVFS for the same performance.

Thermal Controls. Thermal limits ensure protection of the processor from thermal breakdown due to overheating or to stay within the user comfort zone. Though power controls can control heat dissipation, a conservative power limit can result in reduced performance. Sequential applications prefer high frequency cores resulting in high power density. This can result in thermal hotspot although the processor is within its power limit.

Similar to the RAPL mechanism, processors throttle compute units when the chip temperature reaches the critical temperature (temperature beyond which processors can breakdown). Throttling can either be reducing the frequency or the duty cycle (stopping the clock for short periods of 3μs at regular intervals) of the cores or idle thread injection [24]. All compute units are throttled to get the chip back to safer thermal zone. Systems with variable fan speeds can step up fan speeds as they nears critical temperature. These policies are implemented either in BIOS or by the operating system. Processor stay in the throttled state (lower frequency or higher fan speed) for an extended period of time before moving to a normal state to avoid oscillations in and out of the throttling state.

2.2 Motivation

Though processors expose mechanisms to support power and thermal management, they are not sufficient or too coarse-grained to provide process-wide performance guarantees. Current mechanisms are only sufficient to enforce system-level power or thermal limits.

2.2.1 Heterogeneity in Demands

We see at least three reasons on why unequal power distribution among applications is important. First, applications may use power differently to accomplish their goals. Sequential programs may want to power a single core as high as possible, while parallel programs may be faster when spreading power across as many cores as possible. Batch programs may run for long periods at a lower power level, while interactive applications do best with a high-power burst of activity.

Second, within a single system users may have different performance goals for programs. For example, on servers an administrator may want to dedicate power to latency-critical applications at the expense of background batch jobs [15]. On mobile systems, an interactive application may be prioritized for more power and performance than background applications.

Third, hardware itself is becoming heterogeneous, and may have a mix of CPU cores (e.g., ARM's big.LITTLE architecture with in-order and out-of-order cores) or both CPU and GPU cores as in AMD's APUs and Intel's recent processors. The power demand of each type of processing unit can be very different: applications may need more power to use a GPU or big CPU, but receive a super-proportional speedup by doing so. It should be noted that the power requirements of each compute unit varies.

However, the mechanisms supported in current processors for power distribution across different compute units are not sufficient as discussed below.

CPU-GPU Power Distribution. On processors with integrated GPUs, current hardware and software cannot adequately control power across both the GPU and CPU cores. For power management, Intel places them on different power planes thereby enabling individual voltage regulators, and hence frequencies, for each compute units. Through a machine-specific-register (MSR), software can configure the power distribution between CPU and GPU. However, the mechanism does not directly control the power consumption of the two planes: for the same MSR value, the power distribution ratio varies with the number of active CPU cores as well as the total power limit for the processor. Thus, actually controlling power use across both CPU and GPU requires modifying the power distribution (MSR input) when either of the impacting factor changes.

Turbo Boost. Current processors have mechanisms that automatically boost frequencies when there is power/temperature headroom, but this may not always improve perfor-

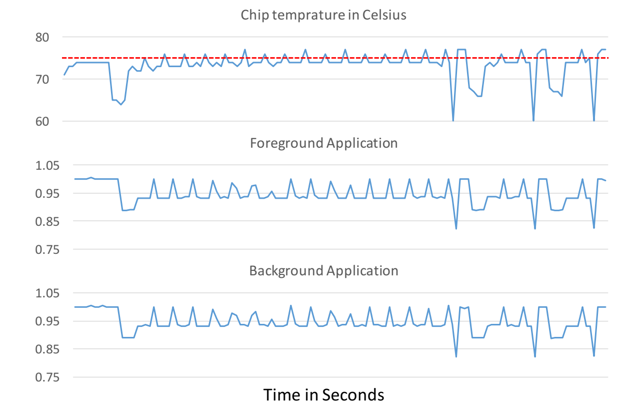

Figure 1. Thermal Interference

mance. Processors from Intel [12], AMD [2] and Samsung [3] boost single-core frequency when neighboring cores are in idle state, indicating there is thermal headroom. However, prior work [16] has shown that such aggressive approach does not always equate to higher performance because not all applications (memory bound) benefit from high frequency. This opportunistic mechanism activates turbo boost whenever thermal headroom is available rather than using it when needed. So, applications that can potentially benefit from the additional frequency cannot leverage turbo boost if no thermal headroom is available.

2.2.2 Reactive Throttling

Current processors enforce thermal limits by throttling the entire processor when the critical temperature is reached. However, activity from low-priority tasks on one core can cause throttling of high-priority tasks on others if the former raise the temperature too high [15]. Figure 1 shows a simple example plotting the performance of a primary and a background application over time when run together. It can be noted that the performance of both applications follow similar pattern of drops over time due to throttling. As the processor temperature reaches the critical temperature, the thermal daemon [34] reduces the frequency of the entire processor. The expected behavior is that primary application performance should not be impacted whereas the background application can be throttled. However, current systems only seek system-wide guarantees and hence throttle the entire system uniformly.

Low priority application can impact other applications even when it is not running alongside them by exhausting the thermal capacity. In cases where a user wants to run a high-priority task overclocked for a short period [26], a background task that already raised the processor to its thermal limit can prevent overclocking, as their is no thermal headroom to further increase frequency. If thermal capacity is treated as a resource by the OS, it can allocate the required thermal capacity to the primary application, so the background application will be forced to run at lower frequency.

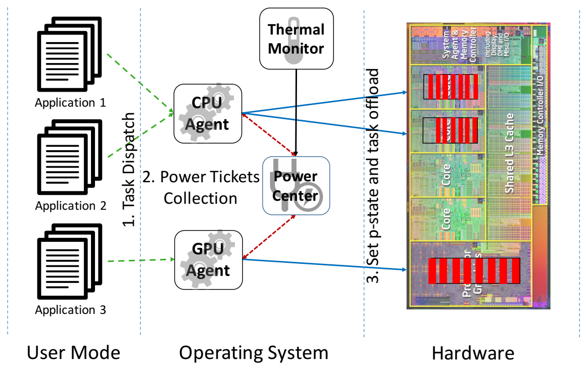

Figure 2. Control flow in Firestorm

3. Design

Firestorm is an extension to Linux that introduces power and thermal awareness in the operating system. The system associates performance requirements of applications with power and thermal requirements. Firestorm enables OS to manage them as primary resources in the system and thus allowing applications to allocate/reserve power and thermal capacity from the system.

Every application in the system receives shares based on their importance (§ 3.1). They get access to resources such as power and thermal capacity based on their shares (§ 3.2 and § 3.3). Any request for computation goes through agents (§ 3.2), compute unit-specific resource managers, that controls performance based on the power and thermal capacity available to the application (based on its shares). The overall flow diagram of Firestorm is shown in figure 2.

The major design goals for Firestorm are:

1. *Power and Thermal capacity as resources.* Just as processors allocate memory and processor time, Firestorm should explicitly control allocation of power and thermal capacity to applications.
2. *Isolation.* Low-priority or malicious applications should not impact other applications due to lack of energy or thermal capacity.
3. *Performance guarantees.* Applications with strong guarantees, such as latency or soft real-time constraints should be guaranteed power thermal capacity to meet deadlines.

A quick note on terminology: we use the words application and program interchangeably, and the term processing units or compute units refers to units such as CPUs, GPUs, and accelerators. A task is a coarse-grained functional block such as a parallel region or function that executes to completion.

3.1 Application Classes

Inspired by the different scheduling classes in Linux, Firestorm divide applications into three different classes. The classification helps to understand the importance of the applications and thus how stringent their requirements are.

The policy decisions — power distribution and preserving thermal capacity — are devised based on the applications classes as can be seen in later sections.

- *Soft Real-Time.* Applications with performance guarantees in terms of SLAs, either periodic or latency limits.
- *Best Effort.* Applications without guarantees that still desire the highest performance based on available resources.
- *Background.* Applications where performance is not a primary goal.

Soft real-time applications are guaranteed power and thermal headroom to meet their performance needs, but require admission control from the system to avoid overcommitting resources. Best-effort applications can fully utilize systems resources as long as soft real-time applications are ensured guaranteed performance. Background or low priority applications are typically not user facing (e.g,. data scrubbing) and do not have stringent performance requirements and should not interfere with applications in other classes. We follow the design principles of previous systems [15] where we give up on background performance to ensure guaranteed or high performance for applications belonging to other classes.

3.2 Power as a Resource

Firestorm focuses on power-limited architectures where all compute units cannot be used at maximum performance within the power budget. The system enforces control over the total power consumed by processing units as well as the power consumed by individual applications. To keep the system within power limits, Firestorm ensures that there is sufficient power *before* allowing an activity to proceed. Note that our focus is power and not energy, although energy-efficient computations are complementary to using Firestorm' mechanisms.

Abstraction. Firestorm abstracts the notion of power in the form of *power tickets* [28, 39] to quantify power as a resource in the system. A power ticket represents the ability to use power: for example, a ticket represents the ability to use one-hundredth of a watt. The difference in power consumption between neighboring frequency levels and duty cycle levels is less than one tenth of a watt for some frequency ranges. We therefore designed the power tickets to be fine grained (rather than a full watt) to capture these differences. The power limit of a processor or system is expressed as a limit on the total number of tickets in use. Thus, dynamically reducing the power limit reduces the number of tickets available. Tickets are managed by one or more *power centers* (one per power socket) that acts as a power source(s) of the system and *agents* request tickets from power center on behalf of the application (discussed under mechanism).

Every power center in the system act as an independent power zone (e.g., one for each socket, one for off-chip GPU). A single centralized power center can become a bottleneck bogged, and having multiple power centers provides the ability to scale with multiple sockets. To use a compute unit, agents contact just the power center hosting unit without affecting other power centers. System-wide power limit can be enforced by ensuring the sum of tickets across all power centers is below the limit. Tickets can also be transferred between centers for long-term power shifting, similar to load balancing of threads across cores and sockets.

Mechanism. In Firestorm, power-consuming portions of the system are controlled by an *agent* that ensures power is available to use the component. An agent is a resource manager for a single type of compute unit and thus every compute unit (CPU, GPU or other accelerators) has its own agent. It also acts as a bridge between applications and compute units similar to a device driver. Applications offload tasks to agent and the agent is responsible for gathering sufficient power tickets from power center on behalf of the application before running the task on the compute unit. The responsibility of an agent is two-fold: (a) It gathers power tickets from the power center on behalf of the application and return the tickets back after task completion. (b) It calculates the number of tickets needed to run a task on the compute unit it manages; more efficient devices require fewer tickets than power-hungry devices, and computations that require less power (e.g., are memory bound) similarly require fewer tickets.

Firestorm employs a pay-before-use model, to ensure the performance of an application is proportional to the power received and also to stay within power limits. Long-running applications in Firestorm can not accumulate power tickets, and applications that are waiting or suspended use no tickets.

Before executing a task, the agent consults its power model (explained in § 4) to determine how many tickets are needed to execute at highest possible p-state (performance level), and requests those tickets. Based on the number of tickets received, the agent configures the hardware to limit its power draw to the amount allocated with help from the power model, such as by lowering frequency/voltage/duty cycle.

Policy. The initial number of power tickets in the power center is set by the administrator, and indicates the power limit for the system. Soft real-time applications reserve power tickets to ensure they have adequate power to execute. The remaining power tickets are shared by applications in best-effort and background class. Every application in the system is assigned shares, and the power center employs a proportional share policy for power distribution across applications. Firestorm also incorporates an additional admission control policy for background class to minimize interference. Background applications execute either when there are no active applications from other classes or a minimum of 50% of total tickets are totally unused (left after allocation to real time and best-effort class) and available in the power center.

Firestorm uses a proportional min-funding mechanism [36] to allocate excess power capacity. If a task requires fewer tickets than an application possesses, the power center re-allocates excess tickets to other applications that could use more power, proportionally to the share of each application. For example, assume application A has shares sufficient to gather 25 power tickets but wants to use a GPU that can consume only 15 tickets even at highest performance. Rather than under-utilizing the extra 10 shares, the power center re-allocates those tickets to other applications proportional to the number of shares they have. Firestorm supports preemption of power tickets when application shares change or a new application (soft real-time or best effort) enters the system. The share allocation is automatically readjusted by the proportional share policy.

3.3 Heat as a Resource

Firestorm takes initial steps toward introducing thermal awareness into operating systems. It avoids or minimizes thermal interference caused by low-priority applications and also allows applications to reserve thermal capacity to ensure guaranteed performance. The above are made possible by promoting thermal capacity as a primary resource in the system.

Abstraction. The thermal capacity of the system is generally defined as the amount of heat needed to raise the system's temperature by one degree [38]. However, to measure the thermal capacity requires knowing the material composition of the heat sink and also properties of the cooling devices (e.g., fan) used. To avoid this complexity, we instead abstract the thermal capacity of the system in terms of the processor chip temperature. The difference between the current chip temperature and the critical temperature is the available thermal capacity of the system. To make it usable, we build a model that predicts the amount of time required for the chip temperature to rise from a start temperature to an end temperature. The model is based on the amount of work done by the processor which is captured in terms of the power consumption of the compute units. More details on the thermal model are discussed later in § 4.2.

Mechanism. Firestorm incorporates a thermal monitor whose goals are to avoid thermal interference and reserve thermal capacity. The first goal is achieved by adding a monitor service that periodically reads the processor temperature sensors to check whether the temperature is nearing the critical temperature. The monitor takes action to reduce temperature through throttling, which lowers the frequency or duty cycle of compute units. This is achieved by notifying the power center to lower the number of power tickets issued to the agents on behalf of applications. The second goal is achieved by exposing a set of interfaces for applications to reserve thermal capacity based on their workload demands.

Policy. The monitor offers two set of throttling policies each targeting different class of applications.

The objective of the *selective throttling* policy is to minimize thermal interference due to low priority applications and thereby trade off the performance of background applications for other applications classes. Firestorm uses selective throttling for best-effort and background applications classes. This policy employs a *reactive approach* in keeping within thermal limits similar to the Linux thermal daemon service [34]: the system only takes action when it reaches a temperature limit. In order to avoid interference caused by background applications, the policy employs a two-stage throttling mechanism based on two temperature limits — a lower background trip temperature and a higher best-effort trip temperature. The lower-stage throttles background applications when the chip temperature reaches the background trip temperature value. The higher-stage gets activated when the chip temperature exceeds the best effort trip temperature value, and throttles applications from both background and best-effort class.

The monitor differs from normal reactive approach by choosing which applications to throttle and thus trades off their performance for others. Throttling is done by reducing the power tickets given to the applications. The monitor conveys two set of information to the power center: (a) application class to be throttled (b) how much throttling as a percentage reduction in power tickets. The effectiveness of throttling depends on how high the chip temperature is compared to the trip values. A large difference demands more throttling and hence a higher reduction in power tickets. It should be noted that best-effort applications also get throttled if throttling background applications is not sufficient to keep within thermal limits. However, background applications get throttled more than best-effort applications.

The *thermal conserve* policy is used for soft real time applications that require guaranteed performance. As discussed before, low priority applications can heat up the processor (exhaust thermal capacity) so much that subsequently scheduled applications cannot run at full performance without getting throttled (lack of thermal capacity). In other words, low-priority applications can affect soft real-time applications by making them miss deadlines/guarantees. For example, in data centers, latency-critical jobs and batch jobs are often scheduled in the same hardware for better utilization. These thermal problems have been shown to be possible in such cases [15]. To avoid this problem, Firestorm allows soft real time applications to reserve the required thermal capacity.

The policy requires knowing in advance the amount of work to be done by the soft real-time application to ensure sufficient thermal capacity is available. The work may be either a high-intensity task running quickly for a medium-intensity task for a longer period. The amount of work is captured by knowing how long an application will run for, and how much power it consumes while it is running, as the power is dispersed as heat. The former value can be predicted by the scheduler from past behavior or provided explicitly by the application as a periodic scheduling requirement. The latter value is obtained by predicting the power consumption of the application through the power model. With these numbers, a thermal model can compute the initial chip temperature that should be set for the application to run unthrottled. In other words, the thermal model computes the minimum thermal capacity needed for the application to run unthrottled. This temperature value is set as the best-effort trip temperature and the background trip temperature is also modified accordingly. Thus, the policy makes sure that even while background or best-effort applications are running, the reserved thermal capacity is available.

3.4 Support for Power Density

Sequential applications deploy all their gathered power on to a single CPU core to run at a high frequency. However, such increased power density can lead to the CPU core becoming a thermal hotspot. Current processors make sure that sufficient thermal headroom is available (other CPU cores are not dissipating any heat) before actually boosting the CPU frequency [2, 3, 12]. This is complementary to the thermal conserve policy where thermal capacity is created rather than preserved. Firestorm includes a new execution object abstraction for sequential applications to create thermal headroom enabling software controlled turbo boost. This is in contrast to the current hardware mechanism (§ 2) where processors activate turbo boost when possible instead of when needed.

Abstraction. The execution object abstraction can be viewed as the combination of execution context with high power density along with required thermal capacity. Firestorm uses execution object to create thermal headroom for sequential applications where all CPUs constituting the execution object except the active CPU are treated as heat sink for the active CPU. The number of CPUs in the execution object is proportional to the thermal headroom required. Sequential applications need to request the kernel for an execution object with the required amount of thermal headroom during its start time. The abstraction is inspired from Chameleon [21] that uses the abstraction to represent an execution context formed from multiple CPU cores in dynamic processors.

Mechanism. The execution object supports two operations: activate and deactivate. Only after gathering sufficient power credits to run at high frequency can an execution object be activated. Activation involves creating thermal headroom and boosting the frequency of the active CPU. This is translated to forcing the constituent CPUs to idle state (except active CPU), increasing the frequency of the active CPU and allowing the sequential application to run on the active CPU. This mechanism is compatible with current processors where Turbo Boost is automatically activated by the processor after execution object activation. Deactivation involves the reverse process where the constituent CPUs are no longer forced to idle state but allowed to run other threads.

Policy. The policy is responsible for making a decision on whether to activate an execution object or not when requested. Naively activating an execution object upon request will prevent other applications from executing since idle threads are forced on other CPUs. The policy strives to balance the requirements for sequential and other applications in the system. Firestorm introduces a configurable knob called *turbo_tax* and it allows the system to to favor sequential or parallel applications or even take a middle ground.

4. Implementation

We implemented Firestorm as an extension to the Linux-4.3.0 kernel. The code changes can be attributed to two major components.

- Power management, including power tickets, agents, power model, frequency balancer and the power sync.
- Thermal management that includes the thermal model and the thermal monitor service along with its policies.

Most code changes were made in the kernel by adding new functionality with a few minor changes to the Linux scheduler to incorporate CPU agents and operations on execution object (activation and deactivation). The total implementation efforts include around 2400 lines of code added to the kernel.

4.1 Power-Aware Scheduling

Firestorm's power-aware scheduling consists of the power center, which manages power tickets; agents that enforce power limits; and a power model for CPUs and GPUs to predict power consumption. We use Intel's clock-modulation feature (also called a duty-cycle mechanism) [13] on CPU cores as a means to reduce performance and thus reducing power, since DVFS cannot be set for each core in our system.

Power Model. We built a simple power model based on linear regression for individual CPU cores through offline analysis. This requires a one-time profiling stage when Firestorm runs for the first time. The profiling stage involves running the SPECCPU 2006 workload suite and measure the power consumption of every workload at different frequencies. We found that integer and floating-point instructions per cycle (IPC and FPC) have high correlation with the CPU power consumption in our test platform. We used the Intel energy performance counter registers to measure the power consumption of the chip. The current model does not support hyperthreading, which introduces additional modeling and control complexities and is left for future work. We built a linear regression model based on these data where the IPC, FPC, frequency and the duty cycle are input to the model and the model predicts the power consumption of the thread running on a CPU core.

The measured chip power includes cores as well as the LLC and other shared structures. To separate out the cost of using a core, we ran a single benchmark on one to four cores and measured the power consumption. We found that power increased linearly, indicating that the per-core cost is the delta in power draw when enabling an additional core. The computed power with zero cores constitutes the LLC and shared structure power.

The CPU agent measures IPC and FPC for every running thread and use the model to predict the maximum power needed to run the application thread. When a thread is context switched in, the agent contacts the power center to request tickets for this maximum power. If the power center does not return the requested tickets, the agent again uses the power model, but this time to determine the highest duty cycle that can be used with the available power.

The GPU agent works similarly. Since most GPU drivers are closed source, we instrument applications to call the GPU agent before task launch to gather power tickets and after task completion to return the tickets. We are still in the process of building a regression based power model for on-chip GPUs. Instead, we run our workloads on the GPU at different frequencies and record the power consumption. The results populate a table for each application, which is later used during experiments to predict the power draw of the same GPU workloads.

Firestorm can operate with different power models for compute units as long as they expose the required interfaces for agents to use them. The list of interfaces to be supported by a power model is given in Table 1.

Frequency Balancer. Intel processors only support a single voltage domain for all CPU cores in a socket. Firestorm must determine an ideal processor frequency to ensure overall system throughput despite supporting heterogeneous performance demands from applications. We observe that most workloads perform better by reducing frequency rather than duty cycle: a thread running at 25% duty cycle at 3 GHz frequency consumes same power as one running at 100% duty cycle at 2.4 GHz frequency, but performs much worse

To avoid over-reliance on duty cycle, Firestorm does not run the processor at the highest frequency requested by an active application. Rather, the balancer of the power center tracks the power tickets given to active threads on all CPU cores. In case of parallel programs, we consider only the request of the primary thread (thread 0) of that program. The balancer identifies the maximum frequency at which each thread could have run with the obtained power tickets. We call this the ideal frequency of the thread. We aggregate the ideal frequency of all active threads to calculate the time-average ideal frequency for all running threads, and select that as the running frequency of the processor. A frequency balancing thread runs periodically every 1 second to recalculate the ideal frequency value. The long interval was picked to amortize the cost of voltage-frequency switching, which is around 50ms.

Applications belonging to the background classes are not considered for these frequency calculation to avoid any interference with other applications. Conversely, if there are applications in the soft real time class, their its frequency request is chosen as the current frequency ignoring any requests from the best-effort class.

Errors in Power Model. In order to cope with the errors in the power model, the power center has a feature called power sync that recalibrates the ratio of power tickets to watts. Errors in power model can either underprice or overprice the power tickets required for a task. The former leads to using fewer power tickets than the actual power consumed, and the latter results in requiring more power tickets for should be needed.

Underpricing can thus cause the system to exceed the power limit even without spending all power tickets. Firestorm avoids this by leveraging the RAPL power limit register. In addition to limiting the total number of power tickets in the system, Firestorm configures the RAPL register to stay within the power budget. Even if power budget is exceeded without consuming all power tickets, the RAPL unit will avoid from exceeding the budget. Overpricing is handled by power sync. For processing units that report power usage, such as Intel Sandy Bridge and later CPUs, the power sync service measures the actual power usage and compares it against the number of tickets in use. If power consumption is below power tickets spent (overpriced), the power sync mechanism creates more power tickets to increase power consumption, and does nothing for the underpriced case since it is taken care by the RAPL budgeting mechanism. It should be noted that the power correction happens for the entire system and not per application.

Interactive services. Interactive threads run for a short period and contribute little to long-term power consumption. Furthermore, the require low latency. Firestorm therefore classifies some threads as interactive services and does not require them to gather power tickets before execution nor run at a reduced duty cycle. All system threads are considered interactive and application threads whose execution time is below their sleep time (maintained by the kernel in their task structures) are also considered interactive.

4.2 Thermal Isolation

The thermal monitor service in Firestorm ensures that no single application overly consumes thermal capacity (i.e., overheats the system) and triggers throttling that hurts the performance of other applications.

Thermal Model. Similar to the power model, Firestorm incorporates a machine-specific thermal model that depends on the type of cooling available and its current state (e.g., current fan speed) in the system. We modeled the temperature trend using a logarithmic regression model where power consumption and time are inputs. It outputs the maximum temperature of the chip at that particular time.

Type	Interfaces	Description
Power Model	`maxpower_for_task (task metrics, frequency)`	Given the task characteristics, return the maximum number of power tickets needed. CPU model in Firestorm takes IPC, FPC, frequency (current processor frequency) as inputs. GPU model takes SM (Streaming multiprocessor) utilization as task metric predict the power.
	`maximum_pstate (task metrics, power)`	The maximum performance state at which a given task can run with the given number of power tickets. For CPUs, Firestorm returns the duty cycle at which the CPU core can run at current processor frequency. For GPU, returns the frequency at which it can run.
	`maximum_frequency (task metrics, power)`	The maximum frequency at which the task can be run using the given power tickets at 100% duty cycle.
Thermal Model	`calculate_temperature (power, time)`	Returns the processor temperature at a particular time with the given amount of work (power).
	`critical_temperature_duration (current temperature, power)`	Returns the time taken to reach the critical temperature given the power consumed by the application.
	`thermal_capacity_needed (power, time)`	Returns the minimum thermal capacity (starting temperature) needed if the application runs for time interval 'time' without exceeding the critical temperature.

Table 1. Power and Thermal Model Interfaces

We use a single profiling stage to measure the thermal performance of the system, which generates a model that can be saved and reused by every application. We assume that the state of the cooling device does not change over the system. A new model has to generated for every state (e.g., different fan speeds) of the cooling device. A model that encompasses all states of the cooling device is left as a future work. The CPU thermal model is generated by running multiple instances of a CPU-intensive workload that heavily uses all functional units. We run these instances at different power limits and record the chip temperature every second using on-chip thermal sensors. The regression model built over the data is

$$temperature = ((a * power) * \log(time)) + (b * power)$$

where a and b are constants. The same model can be used for other applications since the power consumed by them captures the intensity of work done by those applications. While tools like Hotspot [40] offer more accurate models, they require specific details about the processor intricacies, such as the material composition of the heat sink, that make it hard for end-user systems to deploy.

We built a GPU thermal model using similar techniques by running a GPU-intensive workload at different power limits. When multiple workloads run on various compute units (e.g., both CPU and GPU workloads) simultaneously, we observed that the chip temperature is dominated by the highest temperature of the two compute units when the same workloads were run individually. So, we use the maximum of the two model predictions as the chip temperature. Similar to the interfaces for power model, we designed a set of interfaces to be supported by any thermal model integrating with Firestorm, shown in Table 1.

Throttling. The thermal monitor service minimizes thermal interference by reducing performance of low priority applications. The monitor service identifies the class of applications to be throttled based on the trip temperature that is exceeded currently. The monitor informs the power center about the applications class to be throttled. The power center throttles an application by reducing the number of tickets given to an application (agent actually requests for tickets on behalf of the application), and the agent decides the optimal how to use those tickets for maximum performance. The extent of reduction is dependent on how long the chip temperature has been above the trip temperature.

4.3 Execution Objects

Execution objects supports sequential applications by creating thermal headroom for them to compensate for their increased power density. Our design leverages the Linux CFS scheduler [1], which records the virtual runtime for each thread and runs the thread with the lowest runtime next.

An execution object is formed from multiple CPUs where one of them is the active CPU that runs the sequential program and the others are used as heat sinks. In order to represent execution object as an execution context on all the CPUs, virtual threads are created to represent the execution object on all constituent CPUs. Only if all virtual threads manage to occupy the head position in the corresponding run queue, indicating they are next to run, can the activation of the execution object be carried out.

After every execution, the execution runtime, or virtual time spent executing, of the real program thread (on the active CPU) along with the runtime of the virtual threads are updated. If the runtime is not updated, then the virtual threads always gets to stay at the head of the queue. Conversely, if the runtime is updated by a huge value, it will rarely reach the head of the queue. We implement *turbo_tax* as a multiplier of the actual runtime for virtual threads: the runtime is updated as *program_runtime * turbo_tax* where program_runtime is the time an application ran before getting context switched out. If the tax value is less than 1, then

Name	Description
Truecrack [35]	Password cracker, single threaded
Histogram [30]	Finding the frequency of dictionary words in a list of files, OpenCL
x264 [17]	Video Encoder, parallel program
Pbzip [23]	File compression, parallel program
Spec 2006 [10]	Single threaded workload suite

Table 2. Workloads

sequential applications are charged less than the actual time used, which grants them priority ahead of regular threads from other programs. Hence, they get to use neighboring cores as heat sinks more. A tax value greater than 1 does the opposite and favors using cores to run regular threads.

5. Evaluation

We address the following questions in our evaluation: (a) How efficient and flexible is the power distribution infrastructure in Firestorm? (b) Can Firestorm achieve thermal isolation among applications? (c) What is the overhead and accuracy of individual components?

5.1 Experimental Methods.

Platform. We use a Desktop class Intel Ivy Bridge processor i7-6700K with four cores and overclocking enabled. The TDP (Thermal Design Power) for this processor model set by Intel is 91W. It should be noted that the processor does not support per-core DVFS. We use overclocking to show how Firestorm can provide higher sequential performance without being limited by the Intel Turbo Boost mechanism. The native maximum frequency of the CPU is 4 GHz and 4.2 GHZ with Turbo Boost. The GPU has a maximum speed of 1.15 GHz. Overclocking extends the CPU to 4.5 GHz and the GPU to 1.5 GHz. The processor has a single socket and thus runs with a single power center.

We used the RAPL counters for power measurements, on-chip thermal sensors for temperature measurement, Intel's power governor tool [18] for enforcing different power limits on the processor, the Linux thermal daemon service [34] to stay below the critical temperature for the native Linux system. The critical temperature are set to 75°C for the native system. We set the trip temperature values in Firestorm to 70 and 75 for background and best-effort classes respectively, although the trip temperature can be changed dynamically by the thermal conserve policy. We also used the duty cycle mechanism to vary the power of individual cores. We report the average of five runs and variation was below 2% unless stated explicitly.

Workloads. We use the workloads listed in Table 2 for our experiments. The GPU workload are written in OpenCL and built using the Beignet runtime [5] that supports OpenCL interfaces for Intel on-chip GPUs. We measure performance for workloads as throughput: (a) *Truecrack*: words processed per second (b) *x264*: frames processed per second (c) *Pbzip*: data compressed per second (d) *Histogram*: files processed per second.

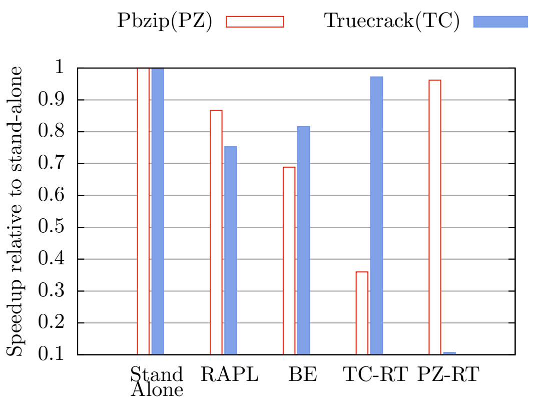

Figure 3. Ticket distribution for applications in soft real time class.

	Balancer Frequency	Truecrack (Desired)	Pbzip (Desired)
TC (standalone)	4.5	4.5	-
PZ (standalone)	3.83	-	3.83
RAPL	3.4	-	-
BE	3.7	4.5	2.9
TC-RT	4.5	4.5	-
PZ-RT	3.7 - 4	-	3.7 - 4

Table 3. Frequency Balancer (Frequencies in GHz)

5.2 Power Management

We analyze whether Firestorm can distribute power to the right set of applications, to ensure guaranteed performance for applications and balance the performance of multiple applications. We set a lower power limit of 30 W (TDP limit is 91W) for all the experiments in this section. The lower limit help us understand the behavior of the system when applications contend for limited power tickets. The limit is enforced by limiting the total number of power tickets to 3000, including tickets used for the LLC.

Sequential and Parallel Applications. This experiment demonstrates Firestorm's ability to balance performance across applications based on their power shares. We ran a single threaded *Truecrack* workload and a parallel *Pbzip* workload at the same time for all configurations (except for standalone) of this experiment. Every application thread runs on its own dedicated core (pbzip was limited to 3 threads). We consider five different configurations: (a) *Standalone*: Applications run standalone in the system within the power limit. (b) *RAPL*: Both applications run in native Linux with power limit enforced through the RAPL counter. (c) *BE*: Both applications are assigned to the best-effort application class with equal shares. (d) *TC_RT*: *Truecrack* is assigned to soft real-time class configured with fixed power tickets (e) *PZ_RT*: Pbzip assigned to the soft real-time class configured with fixed tickets.

The results are shown in the Figure 3 where the performance of applications is normalized to the standalone case.

The *RAPL* and *BE* configurations are comparable since both represent the native configurations of Linux and Firestorm respectively. Under RAPL all cores run at same frequency, so the parallel program achieves higher performance since it uses multiple cores. The sequential program is also made to run at the same frequency (rather than a higher frequency) even though it uses fewer resources (a single core) and therefore performs worse than standalone. Firestorm balances this heterogeneous demand across applications by aggregating individual applications' desired frequency demand. The desired frequency (the standalone frequency given the application's power tickets) and the balancer frequency that Firestorm arrives at are shown in the Table 3. Since both applications receive equal shares under *BE*, *Truecrack in BE* performs better than *Truecrack in RAPL* since it gets to use its power to increase the frequency. On the other hand, RAPL always favors parallel programs by choosing the highest optimal frequency for all cores.

Firestorm provides a RT (soft real time) class for applications demanding guaranteed performance. The applications in RT class receive a guaranteed number of power tickets. We ran separate experiments placing each application in separately the RT class for the last two configurations *TC_RT* and *PZ_RT*, while the remaining application placed as best-effort. *Truecrack* was reserved 1400 power tickets and *Pbzip* with 2400 power tickets. These are the number of power tickets required to achieve performance similar to standalone. The remaining power tickets are used by Firestorm for LLC and the other application. While we determined these shares through manual experimentation, systems that monitor application performance like Heartbeats [11] could be used to set the shares automatically.

In these two configurations, the RT applications achieve within 4% of native performance while companion application suffers. The small performance drop is due to error in power model and fluctuations in the application's characteristics (IPC, FPC) where the reserved tickets is not sufficient to run at full performance. For PZ_RT, the processor run at 3.7 GHz as chosen by the RT application *pbzip*. The power tickets available for *Truecrack* were only sufficient to run at or below 25% duty cycle and hence it performs poorly.

CPU-GPU Applications. This experiment demonstrate how Firestorm distributes power based on application shares across CPUs and GPU, and also its ability to redistribute unused tickets. We ran *Histogram*, a GPU application and *Pbzip*, a CPU application at the same time for all configurations (except for standalone). Both applications belong to best effort class and so the power distribution is dictated by the proportional share policy. We consider five different configurations: (a) *Standalone*: Applications are run standalone in the system with the power limit. (b) *RAPL*: Native Linux with power limit enforced through RAPL counter and the default power ratio setting favors the GPU plane over the CPU power plane. (c) *1:1-Shares*: Both applications receive equal shares. (d) *2:1-Shares*: *Pbzip* receives twice the shares of *Histogram*. (e) *1:2-Shares*: Complement of the previous configuration.

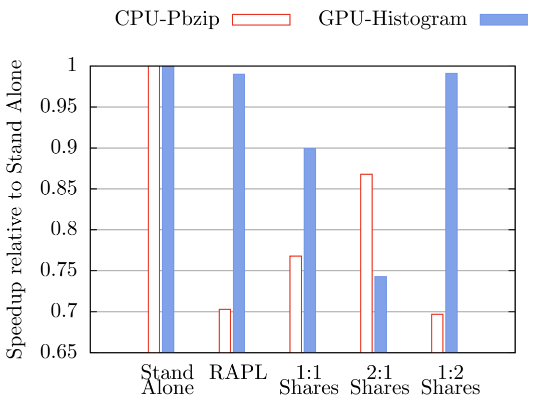

Figure 4. Power ticket distribution in best effort class.

Pbzip (CPU) Tickets	Histogram (GPU) Tickets	Input Ratio	Final Ratio
1736	869	2	1.997
1317	1317	1	1
1061	1579	0.5	0.671

Table 4. Ticket Distribution (Ratio - CPU:GPU)

The results are shown in the Figure 4 where the performance of applications are normalized to the standalone case. As with the previous set of experiments, The *RAPL* and *1:1-Shares* configurations are comparable. The default power ratio across power planes in Linux (*RAPL*) favors the GPU more than the CPU, and as a results *Pbzip* performance drops 29% while *Histogram* drops only 2%. In contrast, Firestorm explicitly allocates equal shares to both applications, which better balances their performance. As shown in the *1:1-Shares*, *Pbzip* drops only 27% and *Histogram* drops 10%.

The performance of applications can be improved by assigning more shares with respect to other applications in the system. This is shown in the last two configurations *1:2-Shares* and *2:1-Shares*. The ticket distribution as per the policy is shown in table 4 in the column *Input ratio*, and the *Output ratio* shows the actual ratio used. The total number of tickets is below the 3000 available since some tickets are spent on the LLC. For *2:1-Shares* and *1:1-Shares*, the ticket ratio follows the input. However, in the case of *1:2-Shares*, *Pbzip* receives more tickets since unused tickets from *Histogram* are reassigned by the power center.

5.3 Thermal Isolation

We evaluate Firestorm's ability to avoid thermal interference among applications to guarantee performance for applications by preserving thermal capacity.

Avoiding Thermal Interference. Current systems do not offer thermal isolation in a shared environment. This experiment shows the ability of Firestorm to isolate applications

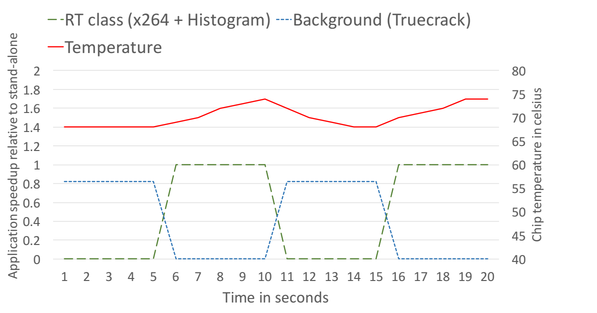

Figure 5. Thermal Conservation: Temperature is plotted with right-side y-axis and speedup follows the left-side y-axis.

from any thermal interference. We ran the same experiment as shown in Figure 1 where two instances of *Truecrack* are run at the same time. One instance is background class and the other is best-effort class. The system was provisioned with sufficient power tickets to run both applications at full performance. However, they cannot be run at full performance without exceeding the critical temperature (75°C).

As shown in figure 1, native linux throttle all applications uniformly: the Linux thermal daemon safeguards the processor from exceeding the critical temperature by reducing temperature for all applications. This results in the performance of both best-effort and background applications dropped by up to 19%. When we perform the same experiment on Firestorm (not shown in the figure), it selectively reduces the performance of background application up to 39% by lowering its duty cycle, while the foreground application is continues to run at full performance (overclocked frequency of 4.5 GHz). The thermal monitor in Firestorm contacts the power center to reduce the power tickets given to background application thus reducing its performance.

Ensuring thermal capacity. This experiment shows the ability of Firestorm to reserve thermal capacity for RT class applications. *Histogram* and *x264* belong to the soft real time class demanding an SLA of 2 files/sec and 5 frames/sec respectively, and *Truecrack* is run as a background application. We interleave RT applications and background application such that each run for 5 seconds. Both RT applications *Histogram and x264* run for 5 seconds followed by *Truecrack* running for the next 5 seconds. We ran 50 such iterations for each configuration and compare the performance of the RT class applications with and without the thermal conserve policy. Sufficient power credits are available in the system to run applications at maximum performance (overclocked frequency of 4.5 GHz).

Figure 5 captures a snapshot of the system for two iterations when the experiment was run using the thermal conserve policy. In native Linux with RAPL, both RT class applications failed to meet their SLAs for more than 50% of the iterations. This occurs because the thermal capacity was exhausted by background application during its 5 second run. The processor is hot when the RT application runs, and as a result gets throttled to a lower speed.

In contrast, with Firestorm's thermal conserve policy, the RT applications run achieve their SLA by running with standalone performance. This is possible since the chip does not overheated while background application runs. Firestorm chooses a throttle limit for the background application based on the thermal requirements of *Histogram* and *x264*. The thermal conserve policy sets a trip temperature of 68°C for the background application and does not allow the background application to exceed that temperature. As a result, the performance of background dropped by 18% in order to satisfy the SLA guarantees of the RT applications.

5.4 Support for Power Density

In addition to distributing power across applications, Firestorm also allows applications to decide how to use power most effectively. Parallel applications spread power across more cores, while sequential applications use power to run a single core as fast as possible, leading to high power density. This can result in causing a thermal hotspot unless sufficient thermal capacity available.

This experiment demonstrates the ability of Firestorm to create thermal headroom for applications. We disabled overclocking support in processor for this experiment and use native Turbo Boost. We want to demonstrate that by creating thermal headroom through execution objects, the processor can use Turbo Boost to execute sequential code at a higher frequency. We use *Truecrack* as the sequential application and *Pbzip* as the parallel application on three cores. The maximum frequency is 4 GHz and the turbo frequency is 4.2 GHz. We ensure sufficient power tickets are available to run at full performance (all cores at 4 GHz).

The different configurations are: (a) *Standalone*: Applications are run standalone in the system. (b) *Native*: Native Linux with both applications running simultaneously. (c) *Tax**: The variants of the configuration are achieved by configuring the *turbo_tax* knob to different input values. It should be noted that lower values prefer sequential over the parallel applications.

The results are shown in the Figure 6 where the performance of applications are normalized to the standalone case. The maximum speedup achievable by the parallel application is 1 because turbo boost will not be activated when multiple cores are active. The *native* case does not show any performance improvement for the sequential application since all cores are active. Tax value of zero would always prioritize the sequential application over others and this would result in sequential application to get same as standalone performance but the parallel performance will be zero. The various *Tax** configurations — 50, 100, 150, 200 — prioritize sequential applications over parallel applications by 2:1, 1:1, 1:1.5, 1:2. The maximum speedup achieved by the sequential application is 5.78% over native. The *turbo_tax* balances the 5.78% of extra sequential performance against 100% of

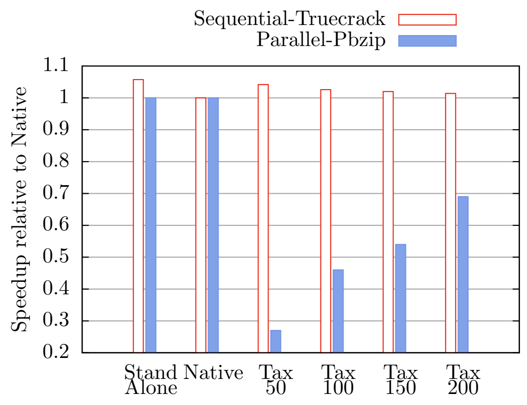

Figure 6. Provisioning for Thermal Headroom.

parallel performance. The various tax values represent the trade-off of opportunity provided by Firestorm: with a high tax rate, the system favors parallel programs as it is hard for *Truecrack* to activate an execution object, so it executes on a single core. With very low tax rates, the system favors running *Truecrack* at a higher frequency, which greatly diminishes *Pbzip* performance. Overall, these results demonstrate how the em turbo_tax configuration setting allows an administrator to balance high sequential against high parallel performance.

5.5 Overhead and Accuracy

We measured the overhead of Firestorm's mechanisms and the accuracy of our performance and thermal models.

Overheads. The primary overhead in Firestorm comes from gathering power tickets from the power center during task dispatch. The overhead comes to 0.75μs for both the CPU and the GPU agents while collecting tickets and 0.4μs while giving it back. It should be noted that the context switch time without the power center is 2μs. Since the CPU agent hooks into the scheduler, the overhead from CPU agent adds upto the context switching time.

Power model accuracy. We measured the power model accuracy by comparing its predicted power consumption for a task against the actual power derived from the RAPL performance counters. We found that the error was close to 16%, largely due to our simplistic model using only IPC and FPC to predict the power consumption.

Thermal model accuracy. We measured the thermal model accuracy by comparing the (temperature) predictions of the model against the actual chip temperature (as measured using on-chip thermal sensors). We ran real workloads under different power limits and comparing the temperatures at different times. The model prediction had an error rate close to 12%.

Model interfaces latency. Bulk of the work to generate the model is done during the system startup time. The interaction of the agent with the model during the system runtime through its interfaces does not cost much. The latency of every interface call is around 0.05μs.

6. Related Work

Thermal Management. There has been great deal of research in this area. Works [4, 6] have compared and analyzed the performance and power aspects of different throttling techniques for thermal management. Currently Firestorm only makes use of duty cycle-based throttling but it can incorporate new throttling mechanisms from these works. Many works [8, 19, 20, 37] have focused on minimizing the system (or chip) temperature through scheduling techniques. A common technique is to migrate computation to a cold resource when the current resource heats up. We think these works are complementary to Firestorm where the scheduling based techniques can be employed using Firestorm's features. There are also works [9] to prevent thermal interference among applications. Firestorm's focus has been to proactively prevent such interference by reserving thermal capacity.

Power Performance Efficiency. Previous works [14, 22, 31] have focused on finding the right processor configuration for a particular application or a mix of applications to achieve the best performance at minimal energy. They also target adapting applications to provide the same performance but reducing the power consumption resulting in high energy efficiency. Firestorm focuses only on mechanisms and policies to distribute power whereas these works focus on optimal setting power setting for a given application.

Power Management. Several systems such as Cinder[28], ECOSystem[39], and Power Containers[29], have been built with a focus on energy management. Firestorm focuses on the similar goal of promoting power as a primary resource and controlling the power use of every application. Firestorm's power ticket abstraction is inspired from these systems. Since the focus of these works is energy, they allow long-running applications to accumulate energy over time, while Firestorm instead grants the ability to execute with a particular power draw at a moment in time.

7. Conclusion

Firestorm allows fine-grained allocation of power to applications. This enables Firestorm to enforce power and thermal limits and to dynamically shift power and thermal capacity between applications and processing units for better efficiency.

As future work, we plan to explore how applications can adapt their workload, such as fidelity of results, in response to varying resource availability and how Firestorm can better support application-level SLAs and achieve overall system efficiency. Also, we would like Firestorm to handle heterogeneous cooling environment where the cooling capacity of the system can change dynamically.

References

[1] CFS Scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[2] AMD Corporation. Amd turbo core technology. http://www.amd.com/en-us/innovations/software-technologies/turbo-core.

[3] Andrei Frumusanu. Early Exynos 8890 Impressions And Full Specifications. http://www.anandtech.com/show/10075/early-exynos-8890-impressions.

[4] P. Bailis, V. J. Reddi, S. Gandhi, D. Brooks, and M. Seltzer. Dimetrodon: Processor-level Preventive Thermal Management via Idle Cycle Injection. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 89–94, 2011. ACM.

[5] BEIGNET. https://01.org/beignet.

[6] J. Donald and M. Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 78–88, 2006. IEEE Computer Society.

[7] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proc. ISCA*, June 2011.

[8] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: Leveraging smt and cmp to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 260–270, 2004.

[9] J. Hasan, A. Jalote, T. N. Vijaykumar, and C. E. Brodley. Heat stroke: power-density-based denial of service in smt. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 166–177, Feb 2005.

[10] J. L. Henning. Spec cpu2006 benchmark descriptions. *Computer Architecture News*, 34(4):1–17, 2006.

[11] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 199–212, 2011. ACM.

[12] Intel Corp. Intel turbo boost technology 2.0. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[13] Intel Corp. Thermal protection and monitoring features: A software perspective. http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/54118.htm, 2005.

[14] J. Li and J. F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 77–87, Feb 2006.

[15] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, 2015.

[16] D. Lo and C. Kozyrakis. Dynamic management of turbomode in modern multi-core chips. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture "'(HPCA)"'*, 2014.

[17] Marth, Erich and Marcus, Guillermo. Parallelization of the x264 encoder using OpenCL. http://li5.ziti.uni-heidelberg.de/x264gpu/.

[18] Martin Dimitrov. Intel Power Governor. https://software.intel.com/en-us/articles/intel-power-governor.

[19] A. Merkel and F. Bellosa. Task activity vectors: A new metric for temperature-aware scheduling. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 1–12, 2008. ACM.

[20] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making scheduling "cool": Temperature-aware workload placement in data centers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 5–5, 2005. USENIX Association.

[21] S. Panneerselvam and M. M. Swift. Chameleon: operating system support for dynamic processors. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 99–110, 2012.

[22] I. Paul, S. Manne, M. Arora, W. L. Bircher, and S. Yalamanchili. Cooperative boosting: Needy versus greedy power management. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[23] Parallel Implementation of bzip2 . http://compression.ca/pbzip2/.

[24] Intel Powerclamp Driver. https://www.kernel.org/doc/Documentation/thermal/intel_powerclamp.txt.

[25] Power Capping Framework. https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt.

[26] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational sprinting. In *Proc. 18th HPCA*, pages 1–12, 2012. IEEE Computer Society.

[27] Intel 64 and IA-32 architectures software developer's manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf, December 2015.

[28] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proc. EuroSys*, 2011.

[29] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power containers: an os facility for fine-grained power and

energy management on multicore servers. In *Proc. 18th ASPLOS*, pages 65–76, 2013.

[30] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: integrating a file system with gpus. In *Proc. 18th ASPLOS*, pages 485–498, 2013.

[31] T. Somu Muthukaruppan, A. Pathania, and T. Mitra. Price theory based power management for heterogeneous multicores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 161–176, 2014.

[32] SpeedFan. `https://en.wikipedia.org/wiki/SpeedFan`.

[33] Surface Book. `https://www.microsoft.com/surface/en-us/devices/surface-book`.

[34] Linux Thermal Daemon. `https://01.org/linux-thermal-daemon`.

[35] Truecrack. `https://code.google.com/p/truecrack/`.

[36] C. A. Waldspurger and W. E. Weihl. An object-oriented framework for modular resource management. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, IWOOOS '96, 1996.

[37] A. Weissel and F. Bellosa. Dynamic thermal management for distributed systems. In *IN PROCEEDINGS OF THE FIRST WORKSHOP ON TEMPERATURE-AWARE COMPUTER SYSTEMS (TACS04)*, 2004.

[38] Wikipedia. Heat capacity. `https://en.wikipedia.org/wiki/Heat_capacity`.

[39] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proc. 10th ASPLOS*, 2002.

[40] R. Zhang, M. R. Stan, and K. Skadron. Hotspot 6.0: Validation, acceleration and extension. Technical Report CS-2015-04, University of Virginia, April 2015.